ChatGPT

Engine with YAML-Defined DAG and Node-Local Evaluation

In this revised approach, the engine reads the DAG structure from a YAML file, and each node type encapsulates its own operation (e.g. addition or multiplication) in its implementation. The engine's role is purely to **parse the YAML**, build the graph, and orchestrate evaluation in the correct order. **All computation logic resides in the node implementations**, not in the engine. This means for an **Add** node, the sum operation appears **only in the Add node's code**, and nowhere in the engine – making the design cleaner and easier to extend.

YAML Input Format for the DAG

We define the DAG in a YAML file where each node is described by its type, a unique id, and any required parameters (like constant value or input references). For example, consider a simple graph with two constants, an addition, and a multiplication:

```yaml
- type: Constant
  id: X
  value: 10
- type: Constant
  id: Y
  value: 5
- type: Add
  id: Sum1
  inputs: [X, Y]
- type: Multiply
  id: Product1
  inputs: [Sum1, Y]
```

Explanation:
- `X` and `Y` are **Constant** nodes with given values.
- `Sum1` is an **Add** node that takes `X` and `Y` as inputs (summing their values).
- `Product1` is a **Multiply** node that takes `Sum1` and `Y` as inputs (multiplying the sum with `Y`).

This YAML will be the input to our engine. By storing the DAG in YAML, we can easily save, load, and re-run computations using pure Rust (no Python required), as the engine will parse this YAML to reconstruct the graph.

Node Trait and Implementations

We define a Rust trait `Node` that all node types will implement. This trait encapsulates all the behaviors needed for evaluation:

- `id(&self)` – returns the node's identifier.
- `get_inputs(&self)` – returns a list of input dependency IDs (empty for constants).
- `is_dirty(&self)` and `set_dirty(&mut self, bool)` – getters/setters for a "dirty" flag indicating if the node needs re-computation.
- `eval(&mut self, nodes: &HashMap<String, Box<dyn Node>>)` – performs the node's computation using the current values of its input nodes.
- `get_value(&self)` – retrieves the node's current output value.
- `set_value(&mut self, value)` – updates the node's value (used primarily for constants).

Using a trait object (`Box<dyn Node>`) allows storing different node types in the same container and calling their `eval` methods polymorphically. Each concrete node struct implements these methods appropriately:

- **ConstantNode:** Stores a fixed `value`. Its `eval` is trivial (no inputs to compute; it simply marks itself not dirty). It overrides `set_value` to allow changing the constant and marking itself dirty.
- **AddNode:** Stores two input IDs and an output. Its `eval` method looks up the values of its two input nodes from the `nodes` map and sets its own `output` as their sum.
- **MultiplyNode:** Similarly, stores two inputs and an output. Its `eval` multiplies the two input values.

Each node type carries its own `dirty` flag. All nodes start as "dirty" (needing computation) when first created or when an input changes. This eliminates the need for a separate `first_run` in the engine – we simply mark all nodes dirty initially and run the evaluation loop.

Below is the implementation of the `Node` trait and concrete node structs:

```rust
use std::collections::HashMap;

// Trait that all node types will implement
trait Node {
    fn id(&self) -> &str;
    fn get_inputs(&self) -> &[String];
    fn is_dirty(&self) -> bool;
    fn set_dirty(&mut self, dirty: bool);
    fn eval(&mut self, nodes: &HashMap<String, Box<dyn Node>>);
    fn get_value(&self) -> i32;
    fn set_value(&mut self, _value: i32) {
        // Default: no-op (only constants override this to update their value)
    }
}

// Constant node: holds a fixed value (no inputs)
```

```rust
struct ConstantNode {
    id: String,
    value: i32,
    dirty: bool,
}
impl Node for ConstantNode {
    fn id(&self) -> &str { &self.id }
    fn get_inputs(&self) -> &[String] {
        // Constant has no input dependencies
        static EMPTY: [String; 0] = [];
        &EMPTY
    }
    fn is_dirty(&self) -> bool { self.dirty }
    fn set_dirty(&mut self, dirty: bool) { self.dirty = dirty; }
    fn eval(&mut self, _nodes: &HashMap<String, Box<dyn Node>>) {
        // Nothing to compute; just mark as not dirty (value is already set)
        self.dirty = false;
    }
    fn get_value(&self) -> i32 { self.value }
    fn set_value(&mut self, value: i32) {
        // Update the constant's value and mark it dirty (needs re-evaluation
for dependents)
        self.value = value;
        self.dirty = true;
    }
}

// Add node: computes the sum of two input nodes
struct AddNode {
    id: String,
    inputs: [String; 2],   // exactly two inputs for addition
    output: i32,           // last computed output
    dirty: bool,
}
impl Node for AddNode {
    fn id(&self) -> &str { &self.id }
    fn get_inputs(&self) -> &[String] { &self.inputs }
    fn is_dirty(&self) -> bool { self.dirty }
    fn set_dirty(&mut self, dirty: bool) { self.dirty = dirty; }
    fn eval(&mut self, nodes: &HashMap<String, Box<dyn Node>>) {
        // Retrieve values of the two input dependencies and sum them
        let a = nodes[&self.inputs[0]].get_value();
        let b = nodes[&self.inputs[1]].get_value();
        self.output = a + b;
        // Mark this node as up-to-date (not dirty)
        self.dirty = false;
    }
    fn get_value(&self) -> i32 { self.output }
```

```
    // No need to override set_value for non-constant nodes
}

// Multiply node: computes the product of two input nodes
struct MultiplyNode {
    id: String,
    inputs: [String; 2],
    output: i32,
    dirty: bool,
}
impl Node for MultiplyNode {
    fn id(&self) -> &str { &self.id }
    fn get_inputs(&self) -> &[String] { &self.inputs }
    fn is_dirty(&self) -> bool { self.dirty }
    fn set_dirty(&mut self, dirty: bool) { self.dirty = dirty; }
    fn eval(&mut self, nodes: &HashMap<String, Box<dyn Node>>) {
        let a = nodes[&self.inputs[0]].get_value();
        let b = nodes[&self.inputs[1]].get_value();
        self.output = a * b;
        self.dirty = false;
    }
    fn get_value(&self) -> i32 { self.output }
}
```

Key Points:

- The **AddNode**'s `eval` performs the sum operation. There is **no summing logic inside the engine** – the engine just calls `node.eval()`. If we add another operation (like Multiply), its logic is confined to the `MultiplyNode` struct. This adheres to the principle that each node's operation is defined in one place only.
- Each node has a `dirty` flag. By setting all nodes to `dirty = true` initially (or whenever an input changes), we can use a single `run` method in the engine for both first-time and subsequent evaluations. No separate `first_run` logic is needed.

Engine Implementation

The `Engine` is responsible for building the nodes from the YAML input and executing the DAG. It holds:

- A map of node ID to `Box<dyn Node>` trait objects (`nodes`), allowing storage of heterogeneous node types.
- A `dependents` map for dirty-flag propagation (maps a node ID to the list of node IDs that depend on it). This helps in marking downstream nodes as dirty when an upstream value changes.

We utilize **Serde** for parsing YAML. First, we define a helper enum `NodeConfig` to deserialize each YAML entry into a Rust structure:

```rust
use serde::Deserialize;

#[derive(Deserialize, Debug)]
#[serde(tag = "type")]  // use the "type" field in YAML to decide which variant
enum NodeConfig {
    Constant { id: String, value: i32 },
    Add      { id: String, inputs: [String; 2] },
    Multiply { id: String, inputs: [String; 2] },
    // ... you can extend with more node types as needed
}
```

The `Engine::from_yaml` function will parse the YAML string into a list of `NodeConfig`, then instantiate the appropriate node structs for each entry:

```rust
use serde_yaml;
use std::collections::HashMap;

struct Engine {
    nodes: HashMap<String, Box<dyn Node>>,       // all nodes in the graph
    dependents: HashMap<String, Vec<String>>,    // mapping from node ID to IDs
of nodes that depend on it
}

impl Engine {
    // Construct an engine (DAG) from a YAML input string
    fn from_yaml(yaml_str: &str) -> Engine {
        // Parse YAML into NodeConfig instances
        let node_configs: Vec<NodeConfig> = serde_yaml::from_str(yaml_str)
            .expect("Failed to parse YAML into node configurations");
        let mut nodes: HashMap<String, Box<dyn Node>> = HashMap::new();

        // Create node objects for each config
        for config in node_configs {
            match config {
                NodeConfig::Constant { id, value } => {
                    // Initialize ConstantNode (mark dirty to compute if needed)
                    let node = ConstantNode { id: id.clone(), value, dirty:
true };
                    nodes.insert(id, Box::new(node));
                }
                NodeConfig::Add { id, inputs } => {
                    let node = AddNode { id: id.clone(), inputs, output: 0,
dirty: true };
                    nodes.insert(id, Box::new(node));
                }
                NodeConfig::Multiply { id, inputs } => {
```

```
                    let node = MultiplyNode { id: id.clone(), inputs, output:
0, dirty: true };
                    nodes.insert(id, Box::new(node));
                }
                // ... handle other NodeConfig variants if added
            }
        }

        // Build the dependents map for propagating changes
        let mut dependents: HashMap<String, Vec<String>> = HashMap::new();
        for node in nodes.values() {
            for input_id in node.get_inputs() {
                dependents.entry(input_id.clone())
                          .or_default()
                          .push(node.id().to_string());
            }
        }

        Engine { nodes, dependents }
    }

    // Evaluate all dirty nodes in the DAG, in the correct order
    fn run(&mut self) {
        let mut progress = true;
        // Iterate until no more nodes were evaluated in an iteration
        while progress {
            progress = false;
            for node in self.nodes.values_mut() {
                if node.is_dirty() {
                    // Check if all inputs have been computed (not dirty)
                    let ready = node.get_inputs().iter().all(|input_id| {
                        self.nodes[input_id].is_dirty() == false
                    });
                    if ready {
                        node.eval(&self.nodes);  // compute the node's value
                        node.set_dirty(false);
                        progress = true;
                    }
                }
            }
        }
        // Detect if any node remains dirty (e.g., due to cycles or missing
inputs)
        let remaining_dirty: Vec<_> = self.nodes.values()
                                         .filter(|n| n.is_dirty())
                                         .map(|n| n.id().to_string())
                                         .collect();
        if !remaining_dirty.is_empty() {
```

```
            eprintln!
("Warning: could not evaluate nodes {:?} (cyclic or missing dependencies)",
remaining_dirty);
        }
    }

    // Update the value of a node (e.g., a constant) and mark its dependents as
dirty
    fn set_node_value(&mut self, node_id: &str, new_value: i32) {
        if let Some(node) = self.nodes.get_mut(node_id) {
            node.set_value(new_value);    // update the node's value if
applicable (only ConstantNode uses this)
            node.set_dirty(true);
            // Recursively mark all dependents as dirty
            if let Some(direct_deps) = self.dependents.get(node_id) {
                for dep_id in direct_deps {
                    self.mark_dirty_recursive(dep_id);
                }
            }
        } else {
            eprintln!("Node `{}` not found in the engine.", node_id);
        }
    }

    // Helper: recursively mark a node and its dependents dirty
    fn mark_dirty_recursive(&mut self, node_id: &str) {
        if let Some(node) = self.nodes.get_mut(node_id) {
            if !node.is_dirty() {
                node.set_dirty(true);
                if let Some(direct_deps) = self.dependents.get(node_id) {
                    for dep_id in direct_deps.clone() {  // clone because we'll
borrow self mutably
                        self.mark_dirty_recursive(&dep_id);
                    }
                }
            }
        }
    }

    // For demonstration: print all node outputs
    fn print_values(&self) {
        for node in self.nodes.values() {
            println!("Node {} = {}", node.id(), node.get_value());
        }
    }
}
```

How the Engine Works:
- **Building the DAG:** `from_yaml` uses Serde to deserialize the YAML into `NodeConfig` variants, then creates actual `ConstantNode`, `AddNode`, etc., and stores them in a map by their `id`. Each new node is initially marked `dirty = true` (meaning it needs computation). The engine also constructs a `dependents` adjacency list to know which nodes depend on a given node (useful for propagating changes).
- **Evaluation Order:** The `run` method loops through nodes, computing those that are marked dirty **only when all their inputs have been computed (not dirty)**. This naturally ensures a topological evaluation order without hard-coding any specific node operations in the engine. The loop repeats until no further progress is made in an iteration. In a proper DAG with no cycles, this will compute all nodes in correct order. If any node remains dirty at the end, the engine logs a warning (e.g., in case of a cycle or missing dependency).
- **Dirty Flags and Re-computation:** Because each node knows its `dirty` status, we use the same `run()` method for the initial computation and all subsequent updates. On first run, all nodes are dirty, so everything gets computed. For updates, the method `set_node_value` can be used to change a node's value (e.g., a constant) and mark it dirty. It then recursively marks any node depending on it as dirty as well. This means only affected parts of the graph will be recomputed on the next `run()`, which is efficient. **No separate first-run logic is needed** – just mark nodes dirty as appropriate and call `run()`.

Notably, the engine does **not** contain any hard-coded logic for summing, multiplying, etc. It simply calls `node.eval()` on each node when it's ready. The actual operation (Sum, Multiply, etc.) is defined **once** in the node's own implementation. This design makes the system easily extensible: adding a new node type means writing a new struct + `Node` trait implementation for it, and adding a case in the YAML parsing – you **do not need to modify the engine's evaluation loop at all**.

Example Usage and Output

Let's walk through using this engine with the example YAML given above. We will:

1. Create the engine from the YAML definition.
2. Run the engine to compute initial values.
3. Print the outputs of all nodes.
4. Change one of the input constants (`Y`) and re-run the engine.
5. Print the outputs again to see the updated results.

Below is a `main` function demonstrating these steps:

```
fn main() {
    // Define the DAG via YAML (could also be read from a file)
    let yaml_data = r#"
- type: Constant
  id: X
  value: 10
- type: Constant
  id: Y
  value: 5
```

```
- type: Add
  id: Sum1
  inputs: [X, Y]
- type: Multiply
  id: Product1
  inputs: [Sum1, Y]
"#;
    // Create the engine from YAML input
    let mut engine = Engine::from_yaml(yaml_data);

    // 1. Initial computation
    engine.run();
    println!("Initial computation:");
    engine.print_values();
    // Expected output:
    // Node X = 10
    // Node Y = 5
    // Node Sum1 = 15       (X + Y = 10 + 5)
    // Node Product1 = 75   (Sum1 * Y = 15 * 5)

    // 2. Modify an input and re-run
    println!("\nAfter changing Y to 7:");
    engine.set_node_value("Y",
7);  // Update Y's value to 7 and mark dependents dirty
    engine.run();                  // Recompute only what's necessary
    engine.print_values();
    // Expected output after update:
    // Node X = 10      (unchanged)
    // Node Y = 7       (updated constant)
    // Node Sum1 = 17   (recomputed: 10 + 7)
    // Node Product1 = 119 (recomputed: 17 * 7)
}
```

Running this program would produce the following output:

```
Initial computation:
Node X = 10
Node Y = 5
Node Sum1 = 15
Node Product1 = 75

After changing Y to 7:
Node X = 10
Node Y = 7
Node Sum1 = 17
Node Product1 = 119
```

As shown, the engine correctly computes the initial values and, after changing `Y` from 5 to 7, re-computes only the affected nodes (`Sum1` and `Product1`). The value of `X` remains the same and was not re-evaluated, demonstrating the dirty-flag mechanism.

Conclusion

This complete example illustrates a robust design for a DAG computation engine in Rust:

- The DAG is defined externally in YAML and loaded at runtime, making it easy to save and reload graphs without Python or other external tools.
- Each node's computation logic is encapsulated in its own implementation of the `Node` trait. The engine does not duplicate any operation logic (no sum or multiply code in the engine).
- A dirty-flag mechanism allows efficient re-evaluation of the graph when inputs change, using the same `run` process as the initial computation.

By following this approach, adding new node types or re-running computations with different inputs becomes straightforward and maintainable. Each new node type requires adding a new struct + trait impl and extending the YAML parsing, but the engine's evaluation loop remains unchanged. This achieves a clean separation of concerns and satisfies the requirement that the operation (e.g., addition) appears in only one place in the code (the node implementation).
